Efficient Locally Weighted Polynomial Regression Predictions

Andrew W. Moore
Robotics Institute and
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
awm@cs.cmu.edu

Jeff Schneider
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
schneide@cs.cmu.edu

Kan Deng
Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
kdeng@cs.cmu.edu

Abstract

Locally weighted polynomial regression (LWPR) is a popular instance-based algorithm for learning continuous non-linear mappings. For more than two or three inputs and for more than a few thousand datapoints the computational expense of predictions is daunting. We discuss drawbacks with previous approaches to dealing with this problem, and present a new algorithm based on a multiresolution search of a quickly-constructible augmented *k*d-tree. Without needing to rebuild the tree, we can make fast predictions with arbitrary local weighting functions, arbitrary kernel widths and arbitrary queries. The paper begins with a new, faster, algorithm for exact LWPR predictions. Next we introduce an approximation that achieves up to a two-orders-of-magnitude speedup with negligible accuracy losses. Increasing a certain approximation parameter achieves greater speedups still, but with a correspondingly larger accuracy degradation. This is nevertheless useful during operations such as the early stages of model selection and locating optima of a fitted surface. We also show how the approximations can permit real-time query-specific optimization of the kernel width. We conclude with a brief discussion of potential extensions for tractable instance-based learning on datasets that are too large to fit in a computer's main memory.

1 Locally Weighted Polynomial Regression

Locally weighted polynomial regression (LWPR) is a form of instance-based (a.k.a memory-based) algorithm for learning continuous non-linear mappings from real-valued input vectors to real-valued output vectors. It is particularly appropriate for learning complex highly non-linear functions of up to about 30 inputs from noisy data. Popularized in the statistics literature in the past decades (Cleveland and Delvin, 1988; Grosse, 1989; Atkeson et al., 1997a) it is enjoying increasing use in applications such as learning robot dynamics (Moore, 1992; Schaal and Atkeson, 1994) and learning process models. Both classical and Bayesian linear regression analysis tools can be extended to work in the locally weighted framework (Hastie and Tibshirani, 1990), providing confidence intervals on predictions, on gradient estimates and on noise estimates—all important when a learned mapping is to be used by a controller (Atkeson et al., 1997b; Schneider, 1997).

Let us review LWPR. We begin with linear regression on one input and one output. Global linear regression (left of Figure 1) finds the line that minimizes the sum squared residuals. If this is represented as

$$\hat{y}(x) = \beta_0 + \beta_1 x \tag{1}$$

then β_0 and β_1 are found that minimize

$$\sum_{k=1}^{N}(y_k - \hat{y}(x_k))^2 = \sum_{k=1}^{N}(y_k - \beta_0 - \beta_1 x_k)^2 \tag{2}$$

During a locally weighted linear regression prediction, a query point, $\mathbf{x}_{\text{query}}$, is supplied. A linear map is constructed, but it is now much more strongly influenced by datapoints that lie close to the query point according to some scaled Euclidean distance metric. This is achieved by (for this prediction only) weighting each datapoint according to its distance to the query: a point very close to the query gets a weight of one and a point far away gets a weight of zero. A common weighting function is Gaussian:

$$\begin{aligned} w_k &= \text{weight of datapoint } \mathbf{x}_k \\ &= \exp(-\text{Distance}^2(\mathbf{x}_k, \mathbf{x}_{\text{query}})/2K^2) \end{aligned}$$

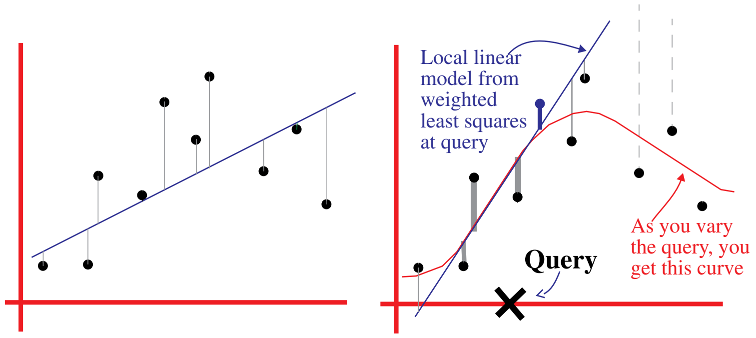

Figure 1: The left graph shows a global linear regression in progress: the sum of squares of the unweighted residuals is minimized. The right graph shows a locally weighted linear regression. The weighted sum of squared residuals is minimized, where the thickness of the lines indicates the strength of the weight.

where the K parameter (called the *kernel width* or *bandwidth* determines how quickly weights decline in value as one moves away from the query. Then instead of finding the line parameters β_0 and β_1 to minimize the global sum of squared residuals, we minimize the *locally weighted sum* of squared residuals:

$$\sum_{i=1}^{N} w_k^2 (y_k - \beta_0 - \beta_1 x_k)^2 \tag{3}$$

The right side of Figure 1 shows the effect. Near the query (marked with an X) large residuals are penalized strongly, but far from the query the penalty is negligible. If the query is moved, then the weights on the datapoints will change and so a different local linear map will be found.

Provided one has a strong stomach for matrix manipulation, these ideas can be easily extended to datasets with many inputs and to local polynomial models instead of linear models. Suppose we wish to estimate a local multivariate polynomial

$$\hat{y}(\mathbf{x}) = \beta_1 t_1(\mathbf{x}) + \beta_2 t_2(\mathbf{x}) + \ldots + \beta_M t_M(\mathbf{x}) \tag{4}$$

around query point, $\mathbf{x}_{\text{query}}$, where t_j is a function that produces the jth term in your polynomial. For example, with two inputs and a quadratic local model we would have $t_1(\mathbf{x}) = 1, t_2(\mathbf{x}) = x_1, t_3(\mathbf{x}) = x_2, t_4(\mathbf{x}) = x_1^2, t_5(\mathbf{x}) = x_1 x_2, t_6(\mathbf{x}) = x_2^2$. Equation 4 can be written more compactly as

$$\hat{y}(\mathbf{x}) = \boldsymbol{\beta}^T \mathbf{t}(\mathbf{x}) \tag{5}$$

where $\mathbf{t}(\mathbf{x})$ is the vector of polynomial terms of the input $\mathbf{x}$: $\mathbf{t}(\mathbf{x}) = (t_1(\mathbf{x}), t_2(\mathbf{x}), \ldots t_M(\mathbf{x}))$. The weight of the kth datapoint is again computed as a decaying function of Euclidean distance between $\mathbf{x}_k$ and $\mathbf{x}_{\text{query}}$. $\boldsymbol{\beta}$ is chosen to minimize

$$\sum_{i=1}^{N} w_k^2 (y_k - \boldsymbol{\beta}^T \mathbf{t}_k)^2 \tag{6}$$

where $\mathbf{t}_k = \mathbf{t}(\mathbf{x}_k)$. Happily, this minimization requires no gradient descent, but can be obtained directly by

$$\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \tag{7}$$

where $\mathbf{X}^T\mathbf{X}$ is an $M \times M$ matrix[1] and $\mathbf{X}^T\mathbf{y}$ is a M-row, 1-column matrix(remember, M is the number of terms) defined thus:

$$(\mathbf{X}^T \mathbf{X})_{ij} = \sum_{k=1}^{N} w_k^2 t_i(\mathbf{x}_k) t_j(\mathbf{x}_k) \tag{8}$$

$$(\mathbf{X}^T \mathbf{y})_i = \sum_{k=1}^{N} w_k^2 t_i(\mathbf{x}_k) y_i \tag{9}$$

More succinctly, we may write:

$$\mathbf{X}^T \mathbf{X} = \sum_{k=1}^{N} w_k^2 \mathbf{t}_k \mathbf{t}_k^T, \mathbf{X}^T \mathbf{y} = \sum_{k=1}^{N} w_k^2 y_k \mathbf{t}_k, \tag{10}$$

2 Efficient LWPR predictions

LWPR is a powerful, flexible tool for fitting multivariate noisy data with non-linearities. But the direct implementation of the above algorithm requires that on every prediction the entire dataset is scanned, weights are computed for all datapoints, and all datapoints have their weighted contribution added into the $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}^T\mathbf{y}$ matrices. With N datapoints and M terms, this means $O(NM^2)$ multiplications to build the matrices and then another $O(M^3)$ operations to compute the $\boldsymbol{\beta}$ vector. Sometimes the computational expense isn't an inconvenience. The devil stick robot of (Schaal and Atkeson, 1994) was able to make 5 predictions a second with 10 inputs, 5 outputs and a thousand

[1] $\mathbf{X}$ itself is an N-row, M-column matrix whose kth row is $w_k \times (t_1(\mathbf{x}_k) \ldots t_M(\mathbf{x}_k))$. $\mathbf{X}$ does not need to be constructed explicitly.

data points using a DSP board. But in cases such as graphing, optimizing over the model, and performing test-set validation of a model, much faster predictions are desirable. And sometimes N (the number of datapoints) may be much larger than a thousand. What then?

Previous researchers have provided three kinds of answers to that question. This paper provides a fourth, described shortly. The first common solution is *editing*, in which only a small subset of all the datapoints are used. This results in an algorithm with rather different properties, in which some information about fine detail is necessarily lost. Another solution is *caching*, in which a multivariate spline model is built when the data is first loaded. (Grosse, 1989) do this with a variable resolution *k*d-tree structure and multilinear interpolation within tree leaves. Unfortunately, continuous interpolation above two dimensions is very expensive in computation and memory. (Quinlan, 1993) also uses a caching method but ignores continuity by storing separate discontinuous linear maps in the leaves. Another downside to caching solutions is that they only record the fitted surface. The scheme used in this paper retains all the information necessary to do a full local regression analysis, providing noise estimates and confidence intervals along with the prediction.

A third solution, and one which *does* retain information, uses a technique called *range-searching* with *kd-trees* (Friedman et al., 1977; Preparata and Shamos, 1985). It is possible to arrange data in such a way that given a query point and a distance, all datapoints within the given distance of the query are returned without needing to search the entire dataset. This works well if there are a small number of dimensions *and* the kernel width is small enough that only a tiny fraction of the datapoints have non-zero weight for a given query. But with more than a few dimensions, the savings for uniformly distributed datasets with fewer than millions of points are disappointing. Much worse, for many datasets, the best kernel width is very broad, meaning that a significant fraction of the data (sometimes all the data) has non-zero weight. In that case, avoiding the zero-weight datapoints is not much help.

In this paper we use the main idea from (Deng and Moore, 1995) in which a multiresolution data structure increased the speed of kernel regression (also known as Locally Weighted Averaging). Here, we extend that method to arbitrary locally weighted polynomials, and give a number of empirical evaluations of the resulting algorithms. We show how an apparently excessive approximation that further reduces prediction times nevertheless gives good performance. We also investigate how these fast predictions can permit prediction-time-optimization of kernel width.

This algorithm builds an enhanced *k*d-tree from the data. Imagine there are two inputs and the datapoints are distributed as in Figure 2 (leftmost figure). Then the root node of the tree records the bounding box of all the data (shown as a rectangle). The root has two children, each of which own half the data and have their own bounding boxes (next part of Figure 2). And in turn they have children. This continues recursively until the leaf nodes, which each contain just one point.

How do we decide which input attribute to split on and where? Unlike decision trees (Breiman et al., 1984; Quinlan, 1983) for induction, the sole purpose of the splits are to increase computational efficiency—not to alter the inductive bias. We do not believe that the choice between the numerous *k*d-tree splitting criteria is critical for this purpose, and so we choose the same "split in the middle of the dimension with the widest spread" method as (Deng and Moore, 1995).

Let *ND* be a node in the *k*d-tree. It records:

- $\mathbf{X}^T\mathbf{X}_{\mathrm{ND}}^{\mathrm{Unweighted}}$: The matrix summing the unweighted data below *ND*.
- $\mathbf{X}^T\mathbf{y}_{\mathrm{ND}}^{\mathrm{Unweighted}}$: The vector summing the unweighted data below *ND*.
- The $\mathbf{y}^T\mathbf{y}$ value for data below the node. This is not needed for predictions as described above, but *is* necessary for computing confidence intervals and noise estimates.

Once the *k*d-tree is built, we can from then on perform cheap computation of weighted $\mathbf{X}^T\mathbf{X}$, $\mathbf{X}^T\mathbf{y}$, and $\mathbf{y}^T\mathbf{y}$ for arbitrary queries, arbitrary (monotonically non-increasing) weighting functions and arbitrary kernel widths. For brevity consider only $\mathbf{X}^T\mathbf{X}$. Given query $\mathbf{x}_{\mathrm{query}}$, suppose we require the weighted $\mathbf{X}^T\mathbf{X}$ matrix for all points below node *ND*.

$$\mathbf{X}^T\mathbf{X}_{\mathrm{ND}}^{\mathrm{Weighted}} = \sum_{k \in BELOW(ND)} w_k^2 \mathbf{t}_k \mathbf{t}_k^T \qquad (11)$$

The obvious method is to ask the two children to compute their own values $\mathbf{X}^T\mathbf{X}_{\mathrm{Left(ND)}}^{\mathrm{Weighted}}$ and $\mathbf{X}^T\mathbf{X}_{\mathrm{Right(ND)}}^{\mathrm{Weighted}}$ and then sum them. If this is all we did there would, of course, be no gain in computation speed as a prediction from the root would still add together $O(N)$ nodes. But sometimes we may be able to cut off a computation at a node. We get our savings if we spot that all the weights below *ND* have near-identical weights given the current query, kernel width and weighting function. This can happen for three reasons:

- All points below *ND* are so far from $\mathbf{x}_{\mathrm{query}}$ that they have zero weight.
- All the points are close together, providing no room for weight variation.
- The weight function varies negligibly over the region below *ND*. For a very wide kernel, a region

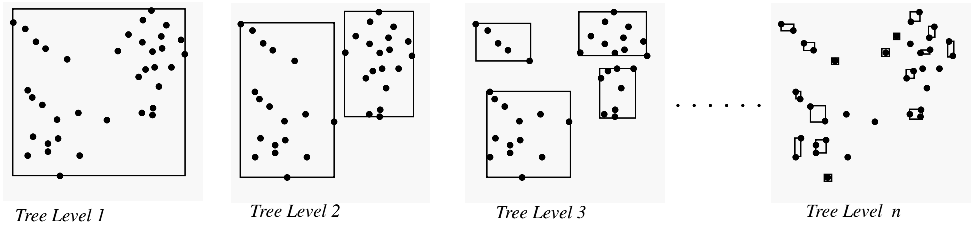

Figure 2: The bounding boxes at increasing depths within the kd-tree.

close to the query may have a constant value of 1, for example.

Computing the maximum variation of weights over all points below node *ND* is easy. We know the location of $\mathbf{x}_{\text{query}}$ and we know the bounding hyper-rectangle of the current node. A simple algorithm costing $O(\text{Number of tree dimensions})$ can compute the shortest and largest possible distances to any point in the node. From these two values, and the assumption that the weight function is non-increasing, we can compute the minimum and maximum possible weights $w_{\min}$ and $w_{\max}$ of any datapoint below node *ND*.

2.1 Exact LWPR using multiresolution

We now have the tools for a simple tree-cutoff rule. When computing the weighted $\mathbf{X}^T\mathbf{X}_{\text{ND}}^{\text{Weighted}}$ matrix, first compute $w_{\min}$ and $w_{\max}$. If they are different, recursively call the weighted $\mathbf{X}^T\mathbf{X}$ computation on the two child nodes and sum the results. If they are identical, write $w = w_{\min} = w_{\max}$ and return

$$\begin{aligned}\mathbf{X}^T\mathbf{X}_{\text{ND}}^{\text{Weighted}} &= \sum_{k\in BELOW(ND)} w_k^2 \mathbf{t}_k \mathbf{t}_k^T \\ &= w^2 \sum_{k\in BELOW(ND)} \mathbf{t}_k \mathbf{t}_k^T \\ &= w^2 \mathbf{X}^T\mathbf{X}_{\text{ND}}^{\text{Unweighted}}\end{aligned}$$

2.2 Approximate LWPR

As we will see the above method can provide large computational savings whilst computing exactly the same local linear model as regular LWPR. But now we will examine how an approximation can further reduce costs. Suppose we are prepared to sacrifice the original weighting function for an approximation that never differs from the original by more than ϵ (Figure 3).

This concession brings tremendous rewards. A simple implementation could then use the following cutoff rule:

If $w_{\max} - w_{\min} \leq 2\epsilon$ then simply return $w^2\mathbf{X}^T\mathbf{X}_{\text{ND}}^{\text{Unweighted}}$ where $w = \frac{1}{2}(w_{\min} + w_{\max})$ because all weights are within $\pm\epsilon$ of w.

This is dangerous. If an LWPR query is far away from any datapoints, then the total sum of weights used in the prediction may be as small as $O(\epsilon)$ and the above approximation may make wildly different predictions than the non-approximate case (especially if thousands of datapoints that should have a weight of 0 are each given a weight of $O(\epsilon)$).

This problem is solved by setting ϵ to be a fraction of the total sum of weights involved in the regression: $\epsilon = \tau\sum_{k=1}^N w_k$ for some small fraction τ. So we would then like to cutoff if and only if $w_{\max} - w_{\min} \leq 2\tau\sum_{k=1}^N w_k$. Of course, we don't know the value of $\sum_{k=1}^N w_k$ before we begin the prediction, and computing it would be a $O(N = \text{number of datapoints})$ operation.

Instead, we estimate a lower bound on $\sum_{k=1}^N w_k$. If, during the computation so far, we have accumulated sum-of-weights W_{SoFar}, and if there are N_{ND} datapoints below node *ND*, and if the minimum weight below *ND* is $w_{\min}$, then

$$W_{\text{SoFar}} + N_{ND}w_{\min} \leq \sum_{k=1}^N w_k \tag{12}$$

These tricks are summarized below. We compute an approximation to the weighted sum $\mathbf{X}^T\mathbf{X}_{\text{ND}}^{\text{Weighted}} = \sum_{k\in BELOW(ND)} w_k^2 \mathbf{t}_k\mathbf{t}_k^T$.

WeightedXtxBelow(ND, W_{SoFar}, τ)

1. Compute $w_{\min}$ and $w_{\max}$. Retrieve N_{ND} and $\mathbf{X}^T\mathbf{X}_{\text{ND}}^{\text{Unweighted}}$ for node *ND*.
2. **If** $(w_{\max} - w_{\min}) \leq 2\tau(W_{\text{SoFar}} + N_{ND}w_{\min})$
3. **Then** return $\mathbf{X}^T\mathbf{X}_{\text{ND}}^{\text{Weighted}} = (\frac{1}{2}(w_{\min} + w_{\max}))^2\mathbf{X}^T\mathbf{X}_{\text{ND}}^{\text{Unweighted}}$.
4. **Else**
   (a) $\mathbf{X}^T\mathbf{X}_{\text{Left(ND)}}^{\text{Weighted}} =$ **WeightedXtxBelow**(Left child of ND, W_{SoFar}, τ)
   (b) Update W_{SoFar} to include the extra weight added by the left child.

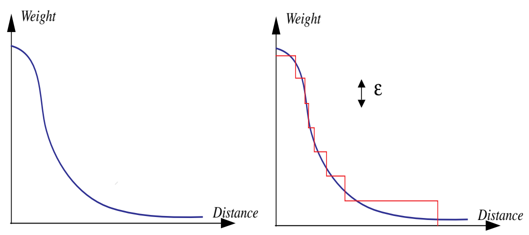

Figure 3: Left side: a weighting function. Right side: an approximation to within tolerance ϵ.

(c) $\mathbf{X}^T\mathbf{X}^{\text{Weighted}}_{\text{Right(ND)}} =$ **WeightedXtxBelow**(Right child of $ND, W_{\text{SoFar}}, \tau$)

(d) Return $\mathbf{X}^T\mathbf{X}^{\text{Weighted}}_{\text{ND}} = \mathbf{X}^T\mathbf{X}^{\text{Weighted}}_{\text{Left(ND)}} + \mathbf{X}^T\mathbf{X}^{\text{Weighted}}_{\text{Right(ND)}}$

Trivial bookkeeping passes the accumulated W_{SoFar} value around.

3 Details

There are several interesting details which we summarize briefly here.

- To ensure numerical stability of this algorithm, all attributes must be pre-scaled to a hypercube centered around the origin.
- The above exposition discussed constructing the $\mathbf{X}^T\mathbf{X}$ matrix. $\mathbf{X}^T\mathbf{y}$ and $\mathbf{y}^T\mathbf{y}$ are constructed in exactly the same way.
- If the function being approximated is linear (or quadratic) in most attributes but nonlinear in a few, one can perform LWPR in which the distance metric only contains the attributes that cause non-linearities. The dimensionality of the kd-tree is only that of the distance metric, not the number of attributes.
- The cost of building the tree is $O(M^2N + N\log N)$. It can be built lazily, (growing on-demand as queries occur) and datapoints can be added in $O(M^2 \times \text{tree depth})$ time, though occasional rebalancing may be needed. The tree occupies $O(M^2N)$ space. Huge memory savings are possible if nodes with fewer than M datapoints are not split, but instead retain the datapoints in a linked list.
- Instead of always searching the left child first it is advantageous to search the node closest to $\mathbf{x}_{\text{query}}$ first. This strengthens the W_{SoFar} bound.
- Ball trees (Omohundro, 1991) play a similar role to a kd-tree used for range searching, but it is possible that a hierarchy of balls, each containing the sufficient statistics of datapoints they contain, could be used beneficially in place of the bounding boxes we used.
- The algorithms have been modified to permit the k nearest neighbors of $\mathbf{x}_{\text{query}}$ to receive a weight of 1 each no matter how far they are from the query. The approximate algorithms use an approximate set of nearest neighbors. This is often useful, but is not discussed further here and is not used in the experiments below.

4 Empirical Evaluation

We evaluated five algorithms for comparison.

Regular	Direct computation of $\mathbf{X}^T\mathbf{X}$ as $\sum_{k=1}^{N} w_k^2 \mathbf{t}_k \mathbf{t}_k^T$.
Regzero	Direct computation of $\mathbf{X}^T\mathbf{X}$ with an obvious and useful tweak. Whenever $w_k = 0$, don't bother with the $O(M^2)$ operation of adding $w_k \mathbf{t}_k \mathbf{t}_k^T$ into the sum.
Tree	The near-exact tree based algorithm (we set $\tau = 10^{-7}$).
Approx	The approximate tree-based algorithm with $\tau = 0.05$.
Fast	A wildly approximate tree-based algorithm with $\tau = 0.5$. This gives an extremely rough approximation to the weight function.

Let's begin with the trivial dataset in Figure 4. Local linear regression is applied with a Gaussian weight function of kernel width 0.48, so that

$$w_k = \exp\frac{-(x_k - \mathbf{x}_{\text{query}})^2}{2 \times 0.48^2} \qquad (13)$$

The middle line of Figure 4 is the predicted value, and the top and bottom lines show the 95% confidence intervals provided by locally weighted regression analysis. The **Tree** algorithm was used. The **Approx** algorithm gives an indistinguishable graph. The **Fast** algorithm gets very similar predictions (Figure 5), but with noticeable small discontinuities. Could these discontinuities cause serious problems for a user trying

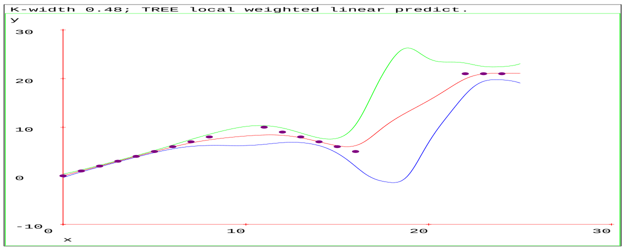

Figure 4: Univariate dataset fitted with locally weighted linear regression using the TREE algorithm.

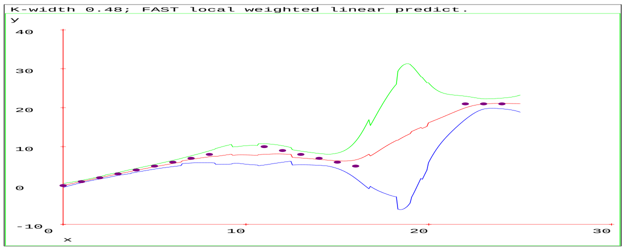

Figure 5: Univariate dataset fitted with locally weighted linear regression using the FAST algorithm.

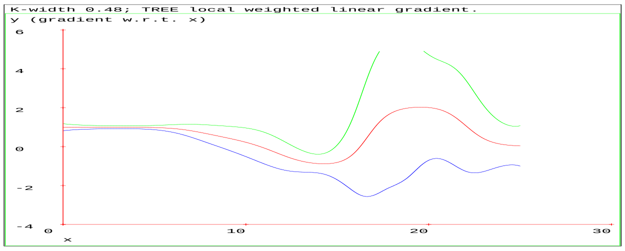

Figure 6: Gradient estimates from the above method.

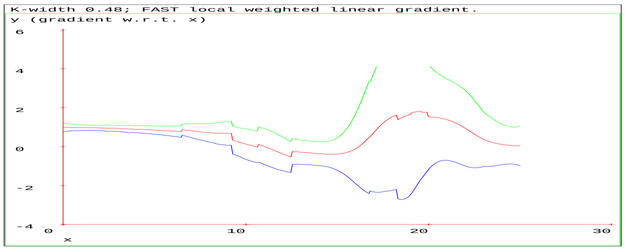

Figure 7: Gradient estimates from the above method.

to estimate derivatives of the surface? Yes, if derivatives are estimated by subtracting close predictions, but derivatives can also be estimated more accurately from the $\boldsymbol{\beta}$ vector identified by the local regression. In this case the derivative is simply the β_1 estimate from Equation 3. The derivatives evaluated the latter way are shown in Figures 6 and 7.

Next we examine prediction on a non-trivial dataset. ABALONE (available from UCI repository) has ten inputs and 4177 datapoints; the task is to predict the number of rings in a shellfish.

In these experiments we removed a hundred datapoints at random as a test-set, and examined each algorithm performing a hundred predictions; all variables were scaled to [0..1], and a kernel width of 0.03 was used. As Table 1 shows, the **Regular** method took almost a second per prediction. **Regzero** saved 20% of that. **Tree** reduced **Regular**'s time by 50%, producing identical predictions (shown by the identical mean absolute errors of **Regular**, **Regzero**, and **Tree**). The **Approx** algorithm gives an eighty-fold saving compared with **Tree**, and the **Fast** algorithm is about three times faster still. What price do **Approx** and **Fast** pay in terms of predictive accuracy? Compare the standard error of the dataset (2.65 if the output variable's mean was always given as the predicted value) against **Tree**'s error of 1.65, **Approx**'s error of 1.67, and **Fast**'s error of 1.71. We notice a small but not insignificant penalty relative to the percentage variance explained.

The above results are from one run on a testset of size 100. Are they representative? Table 2 should reassure the reader, containing averages and confidence intervals from 20 runs with different randomly chosen test-sets. The bottom row shows that the error of **Approx** and **Fast** relative to the **Regular** algorithm is confidently estimated as being small.

Let us examine the algorithms applied to a collection of five UCI-repository datasets and one robot dataset (described in (Atkeson et al., 1997b)). Figure 8 shows results in which all datasets had the same local model: locally weighted linear regression with a kernel width of 0.03 on the unit-scaled input attributes. Figure 9 shows the results on a variety of different local polynomial models described in the caption. The pattern of computational savings without serious accuracy penalties is consistent with our earlier experiment.

There is no space to give the results from numerous other experiments graphing performance against dimensionality, dataset size, kernel width, polynomial type and so forth. A longer technical report is forthcoming.

4.1 Prediction-time optimization of the kernel-width

The above LWPR examples all have fixed kernel widths. There are datasets for which an adaptive kernel-width (dependent on the current $\mathbf{x}_{\text{query}}$) are desirable. At this point two issues arise: the statistical issue of how to evaluate different kernel widths (for example, by the confidence interval width on the resulting prediction, by an estimate of local variance, or by an estimate of local data density) and the computational cost of searching for the best kernel width for our chosen criterion. Here we are interested in the

	Regular	Regzero	Tree	Approx	Fast
Millisecs per predict to build weighted regression matrices	980	800	460	5.7	1.7
Millisecs to solve matrices	3	3	3	3	3
Multiplications	742000	469000	233000	12000	2810
Mean Abs Error	1.65104	1.65104	1.65104	1.67046	1.71381
Low Quartile Abs Err	0.47	0.47	0.47	0.52	0.62
Hi Quartile Abs Err	2.21	2.21	2.21	2.21	2.17

Table 1: Costs and errors predicting the ABALONE dataset.

Algorithm:	Regular	Regzero	Tree	Approx	Fast
Millisecs	98.2 ± 0.25	81.4 ± 0.33	46.8 ± 0.08	0.60 ± 0.016	0.17 ± 0.0036
AbsError Mean	1.534 ± 0.062	1.534 ± 0.062	1.534 ± 0.062	1.536 ± 0.061	1.556 ± 0.063
Excess Error Compared with Regular	0 ± 0	0 ± 0	0 ± 0	0.023 ± 0.034	0.0316 ± 0.032

Table 2: Millisecs to build the weighted regression matrices, errors, and errors relative to **Regular**. 95% confidence intervals are provided based on 20 experiments with 20 testsets.

		Expense	Error
heart.mbl	Regular	42.16	0.27
L40:{9}	RegZero	32.95	0.28
13 inputs	Tree	21.23	0.28
170 datapoints	Approx	18.93	0.28
StdError 0.43	Fast	14.12	0.28
pool.mbl	Regular	34.65	0.63
L40:{9}	RegZero	33.45	0.63
3 inputs	Tree	22.33	0.63
153 datapoints	Approx	4.41	0.63
StdError 2.2140	Fast	0.80	0.62
energy.mbl	Regular	535.87	11.93
L40:{9}	RegZero	484.30	11.93
5 inputs	Tree	323.37	11.93
2344 datapoints	Approx	5.11	15.15
StdError 286.07	Fast	1.10	21.60
abalone.mbl	Regular	964.00	1.65
L40:{9}	RegZero	806.00	1.65
10 inputs	Tree	469.00	1.65
4077 datapoints	Approx	5.80	1.67
StdError 2.66	Fast	1.70	1.71
mpg.mbl	Regular	70.10	1.92
L40:{9}	RegZero	55.18	1.92
9 inputs	Tree	34.35	1.92
292 datapoints	Approx	11.61	1.92
StdError 6.8151	Fast	2.00	1.93
breast-all.mbl	Regular	143.40	0.03
L40:{9}	RegZero	126.18	0.03
9 inputs	Tree	59.88	0.03
599 datapoints	Approx	13.82	0.03
StdError 0.3	Fast	6.21	0.02

Figure 8: Performance on 5 UCI datasets and one robot dataset. HEART and BREAST are classification problems approximated here as a regression on an output of 0 or 1. They threshold their predictions to give a class, and "error" denotes "fraction of testset misclassified". For regression tasks "Error" is "Mean absolute prediction error". In all cases, "Std Error" denotes the error produced by the default rule of "always predict the mean output". Expense is "Millisecs per prediction for building the regression matrices". All tests had a kernel width of 0.03.

		Expense	Error
heart.mbl	Regular	37.86	0.22
A23:{9}-[5]	RegZero	25.84	0.22
13 inputs	Tree	14.32	0.22
170 datapoints	Approx	13.42	0.22
StdError 0.43	Fast	0.50	0.24
pool.mbl	Regular	36.05	0.63
Q50:999	RegZero	35.95	0.63
3 inputs	Tree	25.43	0.63
153 datapoints	Approx	8.12	0.63
StdError 2.2140	Fast	1.20	0.62
energy.mbl	Regular	546.48	6.12
E29:{9}	RegZero	356.12	6.12
5 inputs	Tree	202.29	6.12
2344 datapoints	Approx	25.53	6.03
StdError 286.07	Fast	1.60	7.50
abalone.mbl	Regular	958.90	1.33
L30:999900090-	RegZero	717.34	1.33
10 inputs	Tree	203.91	1.33
4077 datapoints	Approx	2.35	1.33
StdError 2.6611	Fast	1.40	1.34
mpg.mbl	Regular	66.79	1.95
L40:900009090	RegZero	54.18	1.95
9 inputs	Tree	8.41	1.95
292 datapoints	Approx	1.70	1.94
StdError 6.8151	Fast	1.20	1.92
breast-all.mbl	Regular	44.06	0.01
A01:99--99-9-	RegZero	43.96	0.01
9 inputs	Tree	2.20	0.01
599 datapoints	Approx	2.20	0.01
StdError 0.3	Fast	0.50	0.02

Figure 9: Same experiments, but with a variety of LWPR models (each selected by cross-validation). HEART: kernel regression, with kernel width (KW) = 0.015, ignoring one input. POOL: Local Weighted (LW) quadratic regression (thus $M = 10$), KW=0.06. ENERGY: LW Quadratic Regression without cross terms. ABALONE: LW Linear Regression, ignoring one attribute completely and only including 5 attributes in the distance metric. MPG: Uses all attributes but only has three in the distance metric. BREAST: Only uses a subset of five of the ten available inputs.

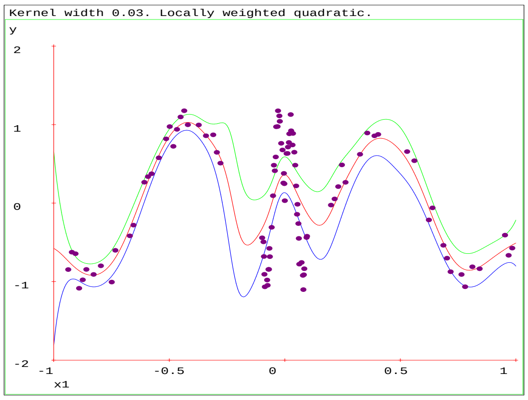

Figure 10: This dataset has 100 points. 50 points have inputs in the range $|\ x\ |< 0.1$, and their outputs are computed as $y = \sin(60 \mid x \mid) + \text{noise}$. 50 points have inputs in the range $0.2 < |x| < 1$ and their outputs are computed as $y = \sin(6 \mid x \mid) + \text{noise}$. Noise std dev is 0.07. It is a dataset crafted to embarrass any fixed-kernel-width method; here you see a local quadratic polynomial with kernel width = 0.03 fitting the outer data well but the inner data poorly. The results in the text use 1000 datapoints with 2-d inputs, and the same separation into two classes of 500 each using the same functions as above (with $|x|$ denoting the distance from x to the origin).

Using Fixed Kernel width		Using Variable K-width		Using Variable K-width, Goal Weight 8.0		
K-width	**Mean Error**	**Goal Weight**	**Mean Error**	**Algorithm**	**Mean Error**	**Millisecs Per Predict**
0.25000	0.41	64	0.19	**Regular**	0.104	2000
0.12500	0.24	32	0.13	**Regzero**	0.104	1400
0.06250	0.24	16	0.11	**Tree**	0.104	395
0.03125	0.22	8	0.10	**Approx**	0.103	181
0.01562	0.29	4	0.10	**Fast**	0.107	165
0.00781	0.37	2	0.11			
0.00391	0.41	1	0.15			
0.00195	0.51	0.5	0.51			

Table 3: Prediction-time optimization of kernel width. Results explained in the text.

computational issue and so we resort to a very simple criterion: the local weight.

Figure 10 shows a 1-input dataset for which a variable kernel width is desirable. In the following experiment we used a two-input dataset constructed in similar spirit. When evaluated on a test-set of 100 points we see that no fixed kernel-width does better than a mean error of 0.20 (Table 3, first two columns). We chose the simplest imaginable adaptive kernel-width prediction algorithm: on each top level prediction make eight inner-loop predictions, with the kernel widths $\{2^{-2}, 2^{-3}, ..2^{-9}\}$. Then choose to predict with the kernel width that produces a weight closest to some fixed goal weight. For dense data a small kernel width will thus be chosen and for sparse data the kernel will be wide. The results are striking. The middle two columns of Table 3 reveal that for a wide range of goal-weights a test-set error of 0.10 is achieved. As the rightmost three columns show, the approximate methods continue to win computationally. This computational efficiency required the ability of the tree based methods to cut off computation even with wide distance metrics.

This experiment can be extended. The search for the goal weights need not build $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}^T\mathbf{y}$ until the goal is found. The search for the best kernel width could binary chop over log(kernel width).

5 Future work

Enormous datasets. A striking feature of the approximate algorithms is that they can often complete locally weighted lookups without ever needing to inspect any individual datapoints. Instead the regression matrices are built up entirely from nodes that summarize the relevant statistics of all the datapoints below them. If datapoints exist at the bottom of the tree without being visited, why put them into main memory? Instead, rarely visited portions of the tree can be transferred to disk until they are queried by extremely local lookups. It is interesting to speculate on extending this to large databases of billions of records of numeric information. There too it may be possible to build a multiresolution structure of records in which parent records recursively record sets of statistics summarizing their descendants. These may be used to greatly speed up queries that ask for statistics of clusters of records local to a given query record. To date we have devised methods for efficiently building such structures of records, and for maintaining the top echelons of the tree in main memory.

SVD-trees. As with other uses of *k*d-trees, a curse of dimensionality remains. If there are hundreds of input attributes in a query, then the hopes for cutoffs during the matrix construction are forlorn: even a hundred levels into the tree some attributes will not have been split upon. We are investigating a new data structure to help combat this problem, called a Singular Value Decomposition Tree. When the set of datapoints in a node is on a linearly dependent subspace, a linear transformation is recorded to map child nodes and all the datapoints to a new lower-dimensional coordinate system.

Acknowledgements

This work was sponsored by a National Science Foundation Career Award to Andrew Moore. Thanks to Mary Soon Lee and the reviewers for valuable comments.

References

Atkeson, C. G., Moore, A. W., and Schaal, S. A. (1997a). Locally Weighted Learning. *AI Review*, 11:11–73.

Atkeson, C. G., Moore, A. W., and Schaal, S. A. (1997b). Locally Weighted Learning for Control. *Accepted for publication in AI Review*, 11:75–113.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth.

Cleveland, W. S. and Delvin, S. J. (1988). Locally Weighted Regression: An Approach to Regression Analysis by Local Fitting. *Journal of the American Statistical Association*, 83(403):596–610.

Deng, K. and Moore, A. W. (1995). Multiresolution Instance-based Learning. In *To appear in proceddings of IJCAI-95*. Morgan Kaufmann.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. on Mathematical Software*, 3(3):209–226.

Grosse, E. (1989). LOESS: Multivariate Smoothing by Moving Least Squares. In C. K. Chul, L. L. S. and Ward, J. D., editors, *Approximation Theory VI*. Academic Press.

Hastie, T. J. and Tibshirani, R. J. (1990). *Generalized additive models*. Chapman and Hall.

Moore, A. W. (1992). Fast, Robust Adaptive Control by Learning only Forward Models. In Moody, J. E., Hanson, S. J., and Lippman, R. P., editors, *Advances in Neural Information Processing Systems 4*. Morgan Kaufmann.

Omohundro, S. M. (1991). Bumptrees for Efficient Function, Constraint, and Classification Learning. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*. Morgan Kaufmann.

Preparata, F. P. and Shamos, M. (1985). *Computational Geometry*. Springer-Verlag.

Quinlan, J. R. (1983). Learning Efficient Classification Procedures and their Application to Chess End Games. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning—An Artificial Intelligence Approach (I)*. Tioga Publishing Company, Palo Alto.

Quinlan, J. R. (1993). Combining Instance-Based and Model-Based Learning. In *Machine Learning: Proceedings of the Tenth International Conference*.

Schaal, S. and Atkeson, C. (1994). Robot Juggling: An Implementation of Memory-based Learning. *Control Systems Magazine*, 14.

Schneider, J. G. (1997). Exploiting Model Uncertainty Estimates for Safe Dynamic Control Learning. In *Neural Information Processing Systems 9, 1996*. Morgan Kaufmann.